Project 2 - Part A : Design Patterns Report

Gage Miller
dgmille2@ncsu.edu
NCSU

Daniel C. Mills
dcmills2@ncsu.edu
NCSU

Daniel E. Mills
demills@ncsu.edu
NCSU

Ian Smart
irsmart@ncsu.edu
NCSU

ABSTRACT

In this paper, we rank ten design patterns according to their utility in the refactoring of individual filters of a six-filter data pipeline into a different programming language. Additionally, we chose a single filter to refactor, utilizing our top three design patterns in the refactoring. At the end of this paper is a reflection on how our design pattern and programming language choices affected the refactoring process.

KEYWORDS

pipeline,monte carlo,design pattern

ACM Reference Format:
Gage Miller, Daniel C. Mills, Daniel E. Mills, and Ian Smart. 2019. Project 2 - Part A : Design Patterns Report. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Our task was to select one filter of the provided data pipeline and refactor it into a different programming language, using three design patterns to guide our implementation. We started by collaboratively working to understand each filter. During this process, we began brainstorming potential ways that filters might be refactored. As a result of our deliberations, we had a handful of design patterns that could be applied to one or several of the filters. Then, we selected a filter and three design patterns based on the practicality of refactoring it using the chosen patterns. These choices then informed our choice of programming language, Typescript. The rest of this paper details these choices, why we made them, and how they affected the refactoring process, for better or worse.

1.1 Pipeline Overview

Brooks Law describes the tendency of late software projects to take even longer to complete when more programmers are added. The "law" was described by its namesake, Fred Brooks, as an "outrageous oversimplification" of how software projects function in the real world [?]. To examine whether Brooks Law does in fact hold true, the provided pipeline generates simulated data describing software project environments and feeds this data into a compartmental model of Brooks Law. The output from the compartmental model is then filtered several times to produce meaningful results. The pipeline as a whole is an example of a Monte Carlo simulation, where a problem is solved by generating randomized input data and then analyzing the results.

The existing pipeline consists of the following six filters, ordered by their sequence in the pipeline:

(1) *monte_carlo*: Generates randomized data describing collaborative software projects
(2) *brooks2*: Runs data through a Brooks Law compartmental model. Calculated dependent variables are attached to the records along with whether they should minimized or optimized.
(3) *dom*: Attaches to every record a *dom* score quantifying the degree to which a record dominates.
(4) *bestrest*: Partitions the records into groups that have low variance

(5) *super*: Attaches the group to which each of the records belong in, ie. best or rest.
(6) *rank*: Lists all the groups in order of how well they select for the best goal

Each filter is an independent program connected by standard Unix-like, anonymous pipes. In practice, this looks like:

```
monte_carlo | brooks2 | dom | bestrest| super | rank
```

1.2 Filter Replacement

Our group decided to replace the *bestrest* filter. *bestrest* begins by sorting the input data set according to its *dom* score, a measure of the degree to which a record Pareto-dominates a random sample of 100 other records. This filter repeatedly partitions the records into two groups such that the expected value of their *dom* score standard deviations is minimized. After each partition, the process is repeated on the records with a *dom* score above the partition boundary, until stopping conditions are satisfied. The records with a *dom* score above the final partition boundary are considered the "best" and those below, the "rest". *bestrest* then appends to every record which of these two groups the record belongs to. The modified data set is then printed to standard output for the next filter, *super*, to take as input.

2 ABSTRACTION RANKINGS

The following subsections detail our considerations of the applicability of design patterns to the refactoring of different filters. Each subsection also includes a short description of our understanding of the pattern. The patterns are ranked from most suitable to least suitable, with the top three being those chosen for our implementation.

2.1 Pipe and Filter

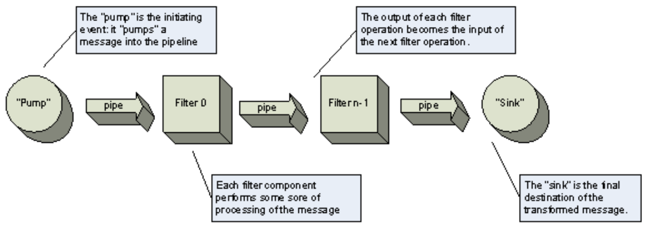

The "pump" is the initiating event: it "pumps" a message into the pipeline
The output of each filter operation becomes the input of the next filter operation.
"Pump"
pipe
Filter 0
pipe
Filter n-1
pipe
"Sink"
Each filter component performs some sore of processing of the message
The "sink" is the final destination of the transformed message.

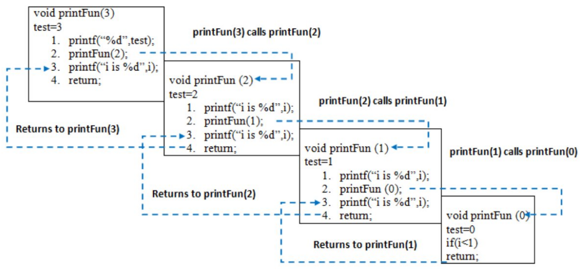
Figure 1: Pipe and Filter Conceptualized. (http://www.rantdriven.com/image.axd?picture=pipe-and-filters-concept.png**).**

A data pipeline is a sequence of "filters", either programs or functions, where the output of each filter is taken as input to the next filter. This pattern is useful when a problem can be broken into independent, sequential sub-problems. By separating a program into independent filters, the filters are typically easier to understand and maintain and they can run concurrently. For the *bestrest* filter, its functionality was separable into three sub-problems:

(1) Sorting the data by "dom" score
(2) Determining how to partition the records into best and rest
(3) Formatting output such that records are labeled according to their partition As a result, this *bestrest* can be separated into three filters, each handling one sub-problem.

2.2 Recursion

Recursion is the idea of a function or algorithm that calls itself during execution. Recursive functions have a wide variety of uses. They can be used for sorting algorithms, iterators for certain types of data structures, and many others. For the pipeline, recursion could be implemented in *bestrest* as a way to sort all of the input data. It could also be implemented in *super* as a way to iterate through all of the data and each subsequent split.



```
void printFun(3)
test=3
    1. printf("%d",test);
    2. printFun(2);
    3. printf("i is %d",i);
    4. return;
```

```
void printFun (2)
test=2
    1. printf("i is %d",i);
    2. printFun(1);
    3. printf("i is %d",i);
    4. return;
```

```
void printFun (1)
test=1
    1. printf("i is %d",i);
    2. printFun (0);
    3. printf("i is %d",i);
    4. return;
```

```
void printFun (0)
test=0
if(i<1)
return;
```

printFun(3) calls printFun(2)
printFun(2) calls printFun(1)
printFun(1) calls printFun(0)
Returns to printFun(3)
Returns to printFun(2)
Returns to printFun(1)

Figure 2: Tracing a Recursive call. (https://www.geeksforgeeks.org/recursion/**).**

2.3 Strategy Pattern

The strategy design pattern allows the user of the program to choose certain algorithms or data structures at runtime. This allows the user to make judgments based on what they know about their input data and how different algorithms can affect efficiency. This design pattern could be used in *bestrest* or *super*, which both require algorithms to sort data. These filters could have the functionality to use command line arguments to let the user choose different sorting algorithms based on which one would be best for their data. Some algorithms have a better expected runtime depending on how the data is already sorted beforehand.

2.4 Adapter Pattern

The adapter design pattern is used to update a program's interface while minimizing necessary changes to client programs using the old interface. In the existing pipeline, each filter, except for *monte_carlo*, formats its output as comma-separated values. *monte_carlo* could be updated to format its output likewise, but this would necessitate that *brooks2*, its subsequent filter, is updated to take comma-separated values as input. To minimize changes to *brooks2* and any other filters that might succeed *monte_carlo*, the Adapter Pattern would prove useful.

2.5 Map Reduce

The map reduce abstraction is a strategy to handle and manipulate large sections of data. It divides the data into smaller chunks, then

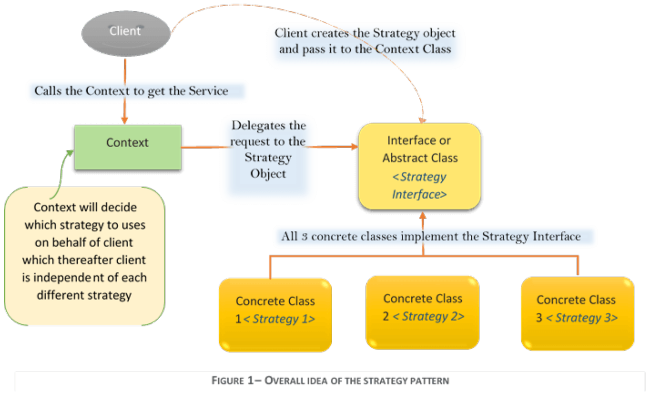

Client
Client creates the Strategy object and pass it to the Context Class
Calls the Context to get the Service
Context
Delegates the request to the Strategy Object
Interface or Abstract Class <Strategy Interface>
Context will decide which strategy to uses on behalf of client which thereafter client is independent of each different strategy
All 3 concrete classes implement the Strategy Interface
Concrete Class 1 < Strategy 1>
Concrete Class 2 < Strategy 2>
Concrete Class 3 < Strategy 3>
FIGURE 1– OVERALL IDEA OF THE STRATEGY PATTERN

Figure 3: Overview of How the Strategy Pattern Works (https://i0.wp.com/www.javagists.com/wp-content/uploads/2017/08/Strategy-Pattern-Overview-2.png?ssl=1).

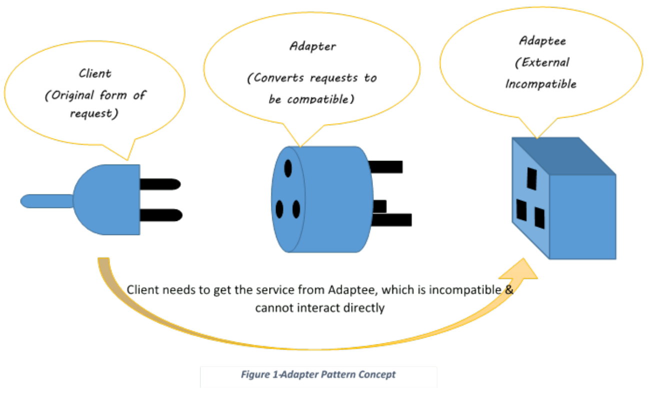

Client (Original form of request)
Adapter (Converts requests to be compatible)
Adaptee (External Incompatible
Client needs to get the service from Adaptee, which is incompatible & cannot interact directly
Figure 1-Adapter Pattern Concept

Figure 4: Example of the Adapter Pattern (https://i2.wp.com/www.javagists.com/wp-content/uploads/2018/01/adapter-Example.png?ssl=1).

maps a function to each chunk. This especially improves efficiency when these functions can run in parallel. Then, a reduce function is applied to bring all the chunks of data back together. This abstraction could be implemented in *monte_carlo* to divide the work of generating attributes into more manageable chunks.

2.6 State Machine

A state machine is a program which keeps track of its state and changes it based on its current state and the current input. It is frequently used in systems which track a certain object as it goes through a series of steps. For example, a bug tracking system, package tracking programs, or customer service requests. In the pipeline, this abstraction could be used in *rank*. Since this filter takes a column, then pushes it through a series of steps, a state machine could be implemented to keep track of each step the column is put through.

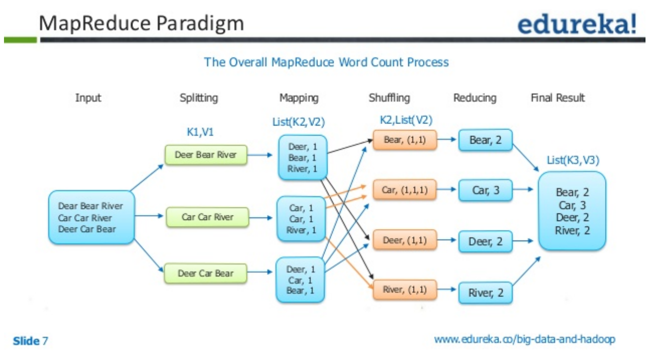


Figure 5: Map Reduce On Reducing Word Count (https://image.slidesharecdn.com/hadoopmr-150613152227-lva1-app6891/95/xml-parsing-with-map-reduce-7-638.jpg?cb=1434209024).

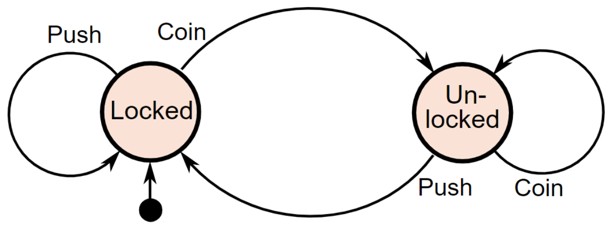


Figure 6: Simple state machine example (https://upload.wikimedia.org/wikipedia/commons/9/9e/Turnstile_state_machine_colored.svg).

2.7 Proxy Pattern

A Proxy is used as a place holder to handle interaction of another object. A proxy routes access to another object in order to provide specific functionality or protection of the real object. A proxy may ensure that access to the real object is done properly or act as a remote device to function like the object. A proxy pattern would be useful in *monte_carlo* to handle attribute data for a generator script. The generator script would be able to accept any generation parameters to produce output, while a proxy file named *"monte_carlo"* would talk to the script and provide the *brooks2* parameters.

2.8 Lambda Calculus

Lambda calculus is the practice of passing lambda bodies to functions to apply them to input. Lambda bodies are just functions or blocks of code which run in a single instance, and can be passed as parameters. This abstraction could be implemented in *dom*. Instead of directly calculating the dom score, it could be done by passing lambda bodies to each row to calculate it. Here's an example of passing a lambda body in Lisp:

```
(mapcar (lambda (x) (* x 2)) '(1 2 3 4))
```

This sends a function which multiplies its parameter by 2 to every value in the list, resulting in the output: `(2 4 6 8)`.

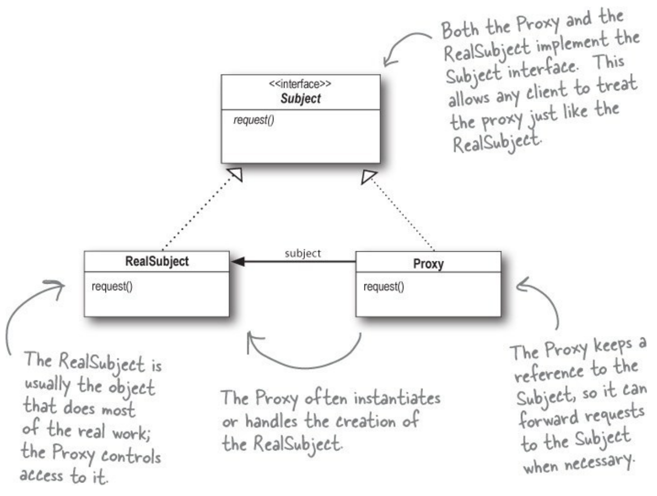

Both the Proxy and the RealSubject implement the Subject interface. This allows any client to treat the proxy just like the RealSubject.
<<interface>>
Subject
request()
RealSubject
request()
subject
Proxy
request()
The RealSubject is usually the object that does most of the real work; the Proxy controls access to it.
The Proxy often instantiates or handles the creation of the RealSubject.
The Proxy keeps a reference to the Subject, so it can forward requests to the Subject when necessary.

Figure 7: Proxy Pattern Explained (https://i2.wp.com/timepasstechies.com/wp-content/uploads/2017/09/proxy.jpg?w=701**).**

2.9 Generator Pattern

A Generator Pattern is used to return a series of outputs but without an array. Instead a generator yields its result and iteratively generates the next result. A generator is an instance of an iterator with a specified start and next value which returns after each call to the generator. The generator pattern is commonly used in random number generation, where the start is a seed value and next is a new random number. This pattern would be very useful in the *monte_carlo* filter, as it would allow a generic generator script to process attribute parameters and generate a new configuration set when calling the generator's next function. This makes it easy to quickly generate thousands of configuration documents with varying parameters and data ranges.

2.10 Macros

Macros are used to mapping a snippet of code to series of characters. Then, when the compiler encounters that series of characters, it replaces the series of characters with the snippet of code. This helps make code easier to read and allows programmers to write less code when repeatedly performing the same operation. Macros are used all over the place, especially in languages like Lisp, C, Clojure, Elixr, and Prolog. A few specific examples of where macros might be used includes assigning a large number to a short name (such as PI to represent 3.1415...), finding the average of two numbers, or even creating a macro which includes multiple macros. This pattern would be very useful in the rank filter, as it requires multiple uses of long numbers and several identical function calls. Those could be cut and replaced with macros, which would make the filter infinitely easier to understand. Here's an example of defining and using a macro in Lisp:

```
(defmacro mulsq (x y) "multiplies the squares" ‘(*
(* ,a ,a) (* ,b ,b)))
```

The macro can now be used to multiply the squares of two numbers, for example `(mulsq 4 5)` results in 320.

3 EPILOGUE

At the start we had a hard time deciding on which abstraction to choose with which filter. Our initial plan was to do the *monte_carlo* filter with a map reduce abstraction to quickly generate the configurations for brooks2. We would use multiple threads or virtual cores to generate output for each attribute and combine the returned value to a list. We also considered using multiple threads that would generate separate attribute lists that would compile to one massive attribute list and pushed to standard output. We soon realized that this solution would be overkill, and using any multi-threaded solution would require artificial timing buffers to replicate the map reduce abstraction.

Next we tried to work with some other simple abstractions with the *monte_carlo* filter and work with a general configuration generator. The general configuration builder would be able to take any number of attributes where as the current *monte_carlo* is built only for the *brooks2* attributes. This new configuration builder would enable us to define different types of data ranges such as continuous or discrete random number generation. We would also be able to add other types of attributes such as Boolean values or lists. This would use both the proxy and generator patterns with the potential for lambda calculus to pass in specific generation functions. However most of the lambda functions would not be useful for the *monte_carlo* generation, and would be ineffective for this project.

Finally, we decided to work with the *bestrest* filter. This filter partitions the data into groups with low variance. The data is first sorted by each row's dom score. Next, the filter uses the variance of each row to find out where to split the data. Once this is calculated, program then divides the data based on variance, one group being marked the *best*, with the other being the *rest*. Output is then generated with the last column displaying the range of the best and rest.

Given the step-by-step nature of this filter, we realized it was a perfect opportunity to implement the *pipe-and-filter* design pattern. We decided to divide this filter into three filters which each handle a smaller task. The first filter, *sortlastcol* handles the sorting of the data. Its input data is the output from the dom filter, which is a series of comma-separated values. The last value of each row is the dom score, so the *sortlastcol* filter sorts all the rows of data by the value in the last column.

This first filter was also an opportunity to implement a few more abstractions. First, given that the goal is to sort large quantities of data, we decided to use a recursive sorting algorithm to perform this task efficiently. The default sorting algorithm for this filter is mergeSort, which recursively divides the data in half, then merges each division together to make it sorted. Second, we saw the opportunity to implement the strategy pattern. This is a design pattern which allows users to select which algorithm is used at runtime. Given that some sorting algorithms can be more efficient depending on the input data, it makes sense to allow the users to choose which algorithm sorts their data. We added functionality to allow the user to enter command line arguments to choose their sorting algorithm: -m for mergeSort (default), -b for bubbleSort, -s for selectionSort, or

Table 1: Bonus

Bonus	Stars	Marks
TypeScript	3	2
Pipe and Filter	2	1
bestrest	2	1
Project	NA	10

if they want to be especially efficient, -bogo for bogoSort.

The next filter is *argmin*. This filter handles the calculation of the index on which to split the records into "best" and "rest", according to their *dom* values. This is accomplished by iteratively splitting the records into partitions such that the expected value of the partitions' standard deviations of *dom* values are minimized. After a split, the process is repeated for the records above the partition boundary. If a part has less than a minimum allowed *dom* value range or number of records, the iteration stops. The records above the last partition boundary are considered the "best" and those below the "rest". The records are then labelled according to the partition they belong to and printed. The only new abstractions utilized within *argmin* were several uses of lambda functions mapped across the input. In retrospect, recursion could have replaced the iterative parts of calculating the split values, as in the original *bestrest.*

The last filter in our pipe is *Pipe3*. *Pipe3* modifies the last line to include the range of the cut from best to rest. It also displays all cuts made to stderr as a visual aid. The input into *Pipe3* is formatted with the next cut on the last column of each line.*Pipe3* uses recursion to print the current cut to standard out and then continues until the last cut is found to mark the whole list. After marking the last column with the appropriate ranges, *Pipe3* iterates through the list and prints to stdout.

4 EXPECTED GRADE

Completion of the project with expected output and meeting all requirements should net us 10 marks. Doing the *bestrest* filter, which is a 2 star filter, would give us an additional 1 mark. We choose to program in TypeScript, a 3 star language, which would give us 2 more marks. Finally we used the pipe and filter abstraction which gives us an additional 1 mark. In total we should have 14 marks for the project. 1